4. Employee Management System

Understanding Array Representation

Arrays in Memory:

Contiguous Memory Allocation: Arrays are stored in contiguous memory locations. This means that if the array starts at memory address x, and each element occupies k bytes, the elements of the array are located at addresses x, x + k, x + 2k, and so on.

Advantages:

- **Fast Access**: Arrays provide O(1) time complexity for accessing elements by index. This means you can retrieve any element directly if you know its index.
- **Memory Efficiency**: Arrays have a fixed size, which allows for efficient memory allocation and deallocation since the memory is allocated in a single block.
- **Cache-Friendly**: Due to contiguous memory allocation, arrays are more cache-friendly, leading to better performance in terms of access speed. Sequential access of array elements benefits from spatial locality.

Setup And Implementation:

```
/*
 * To change this license header, choose License Headers in Project Properties.
 * To change this template file, choose Tools | Templates
 * and open the template in the editor.
 */
package EmployeeManagementSystem;


/**
 *
 * @author Aishwarya
 */
import java.util.Scanner;


class Employee {
    private int employeeId;
    private String name;
    private String position;
    private double salary;

    public Employee(int employeeId, String name, String position, double salary) {
        this.employeeId = employeeId;
        this.name = name;
        this.position = position;
        this.salary = salary;
    }

    public int getEmployeeId() {
        return employeeId;
    }

    public String getName() {
        return name;
    }

    public String getPosition() {
        return position;
    }

    public double getSalary() {
        return salary;
    }

    @Override
    public String toString() {
        return "Employee ID: " + employeeId + ", Name: " + name + ", Position: " + position + ", Salary: $" + salary;
    }
}

public class EmployeeManagementSystem {
    private Employee[] employees;
    private int employeeCount;

    public EmployeeManagementSystem(int capacity) {
        employees = new Employee[capacity];
        employeeCount = 0;
    }

    public void addEmployee(Employee employee) {
        if (employeeCount < employees.length) {
            employees[employeeCount] = employee;
            employeeCount++;
            System.out.println("Employee added successfully.");
        } else {
            System.out.println("Employee array is full. Cannot add more employees.");
        }
    }

    public Employee searchEmployee(int employeeId) {
        for (int i = 0; i < employeeCount; i++) {
            if (employees[i].getEmployeeId() == employeeId) {
                return employees[i];
            }
        }
        return null;
    }
```

```
public void traverseEmployees() {
    if (employeeCount == 0) {
        System.out.println("No employees found.");
        return;
    }
    for (int i = 0; i < employeeCount; i++) {
        System.out.println(employees[i]);
    }
}

public boolean deleteEmployee(int employeeId) {
    for (int i = 0; i < employeeCount; i++) {
        if (employees[i].getEmployeeId() == employeeId) {
            for (int j = i; j < employeeCount - 1; j++) {
                employees[j] = employees[j + 1];
            }
            employees[employeeCount - 1] = null;
            employeeCount--;
            return true;
        }
    }
    return false;
}

public static void main(String[] args) {
    EmployeeManagementSystem ems = new EmployeeManagementSystem(10);
    Scanner scanner = new Scanner(System.in);

    while (true) {
        System.out.println("\nEmployee Management System");
        System.out.println("1. Add Employee");
        System.out.println("2. Search Employee");
        System.out.println("3. Traverse Employees");
        System.out.println("4. Delete Employee");
        System.out.println("5. Exit");
        System.out.print("Choose an option: ");
        int choice = scanner.nextInt();

        switch (choice) {
            case 1:
                System.out.print("Enter Employee ID: ");
                int employeeId = scanner.nextInt();
                scanner.nextLine();
                System.out.print("Enter Employee Name: ");
                String name = scanner.nextLine();
                System.out.print("Enter Employee Position: ");
                String position = scanner.nextLine();
                System.out.print("Enter Employee Salary: ");
                double salary = scanner.nextDouble();
                Employee newEmployee = new Employee(employeeId, name, position, salary);
                ems.addEmployee(newEmployee);
                break;

            case 2:
                System.out.print("Enter Employee ID to search: ");
                employeeId = scanner.nextInt();
                Employee foundEmployee = ems.searchEmployee(employeeId);
                if (foundEmployee != null) {
                    System.out.println("Found Employee: " + foundEmployee);
                } else {
                    System.out.println("Employee not found.");
                    }
                    break;

                case 3:
                    System.out.println("Current Employees:");
                    ems.traverseEmployees();
                    break;

                case 4:
                    System.out.print("Enter Employee ID to delete: ");
                    employeeId = scanner.nextInt();
                    if (ems.deleteEmployee(employeeId)) {
                        System.out.println("Employee deleted successfully.");
                    } else {
                        System.out.println("Employee not found.");
                    }
                    break;

                case 5:
                    System.out.println("Exiting...");
                    scanner.close();
                    return;

                default:
                    System.out.println("Invalid option. Please try again.");
            }
        }
    }
}
```

Analysis: Time Complexity and Limitations of Arrays

Time Complexity:

- **Add (at the end)**: O(1) if there's space. O(n) if resizing is needed (dynamic arrays).
- **Search**: O(n) for unsorted arrays (linear search), O(log n) for sorted arrays (binary search).
- **Traverse**: O(n), as each element is accessed once.
- **Delete**: O(n) in the worst case, as elements may need to be shifted to fill the gap.

Limitations of Arrays:

- Once an array is allocated, its size cannot be changed**(fixed size).** If you need a dynamically sized collection, you might need to use a dynamic array (e.g., ArrayList in Java) or another data structure like a linked list.
- Inserting or deleting elements (other than at the end) requires shifting elements, leading to **O(n)** time complexity. This can be **inefficient** for large datasets.
- If the array is not fully utilized, it leads to **wasted memory**. Conversely, if the array needs to grow, it requires copying elements to a new, larger array, which is time-consuming.
- Arrays are **not suitable** for scenarios where frequent **random insertions and deletions** are required **(Sequential Access)** . Linked lists or other dynamic data structures might be more appropriate in such cases.

When to Use Arrays:

- When the size of the dataset is known and fixed.
- When fast access to elements by index is required.
- When memory overhead needs to be minimized.
- When the operations are mostly traversals or accessing elements by index, and not frequent insertions or deletion